Using MATLAB to solve differential equations numerically

Morten Brøns
Department of Mathematics
Technical University of Denmark

September 1998

Unfortunately, the analytical tool-box for understanding nonlinear differential equations which we develop in this course is far from complete. The good news is that with the present day computer power it is possible to compute numerically approximate solutions to the equations. The purpose of the present chapter is to give a quick introduction to this subject in the framework of the programming language MATLAB. We will focus on practical matters and readers interested in numerical analysis as a mathematical subject should look elsewhere.

In the G-databar at DTU, MATLAB can be accessed either by typing `matlab` at a terminal prompt or by pressing the middle mouse button when the cursor points at the terminal background. A small menu pops up, and MATLAB can be chosen under the item *mathematics*. In any case, a terminal window with the MATLAB prompt “>>” results, and if this is your first try you are advised to type `demo` to get a demonstration of the program's capabilities. Also, the command `helpwin` gives an interface to the on-line help system.

You probably want to save all material for this course in a directory, say, `dynamics`; having created this the matlab command `cd dynamics` will make it the working directory.

Before anything can be done with a differential equation, a so-called `.m`-file defining the right hand side must be created. As an example, consider the van der pol equation

$$\dot{x} = y, \;\; \dot{y} = (1 - x^2)y - x. \tag{1}$$

Open your favorite text editor, enter the following

```
function xdot = vdp1(t,x)
xdot = zeros(2,1);
xdot(1) = x(2);
xdot(2) = (1-x(1)^2)*x(2) - x(1);
```

and save it as `vdp1.m`.

The first line defines a function `vdp1`, a function of time `t` and the state vector `x`, which returns the value of the right hand side of the system (1). Since the system is autonomous, time does not appear explicitly. The next line initialises the function to be a matrix with 2 rows and 1 column – i.e. a column vector – consisting of zeroes. The two final lines then compute the two coordinates of the right hand side. The lines end with a semi-colon to prevent the result from being printed when the function is called.

To solve this equation numerically, type in the MATLAB command window

```
>> [t,x] = ode45('vdp1', [0 20], [1 0]);
```

(except for the prompt generated by the computer, of course). This invokes the Runge-Kutta solver `ode45` with the differential equation defined by the file `vdp1.m`. The equation is solved on the time interval $t \in [0,20]$ with initial condition $(x(1),x(2)) = (1,0)$. The solver obtains results at times stored in the one-column vector `t` with the corresponding solution in the two-column matrix `x`. The solver has an error control and adjusts the time step to try to make the error within a certain bound. Hence, the number of steps taken in not known in advance. Check the actual size by e.g. `size(x)`. Further information on the solver can be obtained by `help ode45`.

You can plot the variables as a function of time using

```
>> plot(t,x(:,1),'r',t,x(:,2),'b-.')
```

A new window opens with the plot. Here `x(:,1)` means everything in the first column in `x` and `'r'` gives a red curve. The second column is shown in blue with a dash-dotted line. Use `help plot` to see how to control the plot.

A plot in the phase plane of the solution may be obtained by

```
plot(x(:,1),x(:,2))
```

See figure 1.

Note, that in the MATLAB command window you can scroll through previous commands by using the up and down arrow keys. This is most useful for editing and reusing previous commands.

The Runge-Kutta method used above is a good choice for a standard solver. However, for some systems of differential equations the error control will force the solver to take extremely small steps, and more advanced methods are needed. Such systems are called *stiff*, and this occurs typically when the dynamics involve both slow and fast changes. Methods for stiff problems are generally implicit, that is, in each time step a non-linear set of algebraic equations must be solved. This is time-consuming, but the advantage is that much larger steps can be taken and the overall performance is significantly improved.

To illustrate stiffness, consider a modified van der Pol equation

$$\dot{x} = y, \;\; \dot{y} = 1000(1-x^2)y - x. \tag{2}$$

Make a file `vdp1000.m` with the following contents:

```
function xdot = vdp1000(t,x)
xdot = zeros(2,1);
xdot(1) = x(2);
xdot(2) = 1000*(1-x(1)^2)*x(2) - x(1);
```

and solve as before but with a very short time interval $t \in [0,1]$:

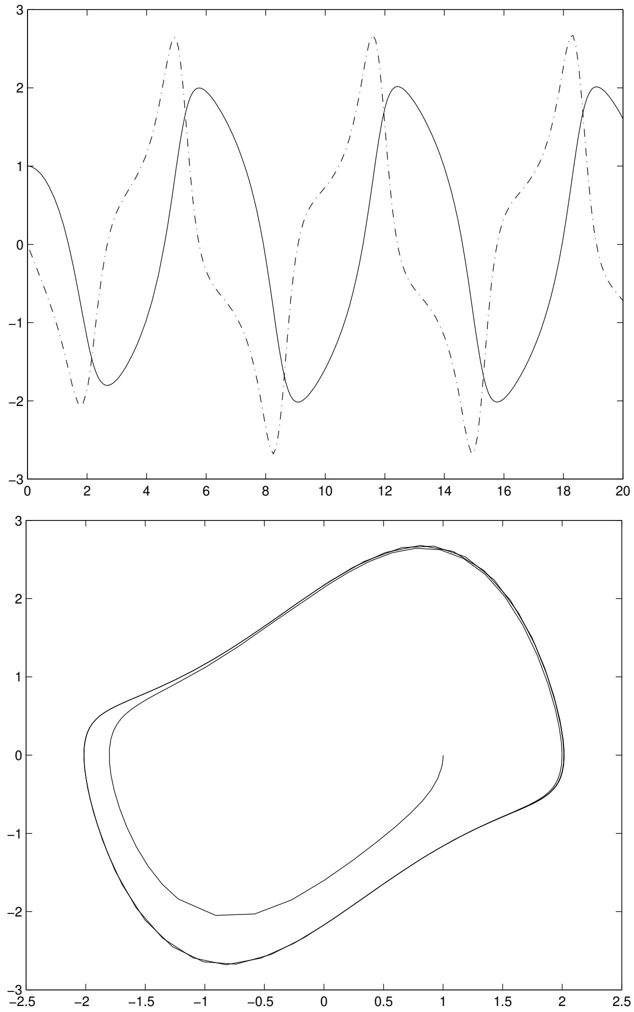

FIGURE 1: MATLAB plots of a solution to the van der Pol equation (1). Top: Time traces. Bottom: Phase plane.

```
>> [t,x] = ode45('vdp1000', [0 1], [1 0]);
```

How many steps where taken?

```
>> size(t)

ans =

        3033            1
```

The array t has 3033 rows (and 1 column) so this is the number of steps. MATLAB also has an implicit solver for stiff systems, ode15s, which may be used as follows:

```
>> [t,x] = ode15s('vdp1000', [0 1], [1 0]);
>> size(t)
```

```
ans =

   165     1
```

Here, only 165 steps was needed and the computation was performed much faster.

In fact, the time interval considered is much too short to be of interest. These commands give the plots in figure 2:

```
>> [t,x] = ode15s('vdp1000', [0 3000], [1 0]);
>> plot(t,x(:,1))
>> plot(x(:,1),x(:,2))
```

Note the long period of the limit cycle compared to figure 1, the different scales of the two variables, and the slow-fast behavior which is the source of the stiffness of the system.

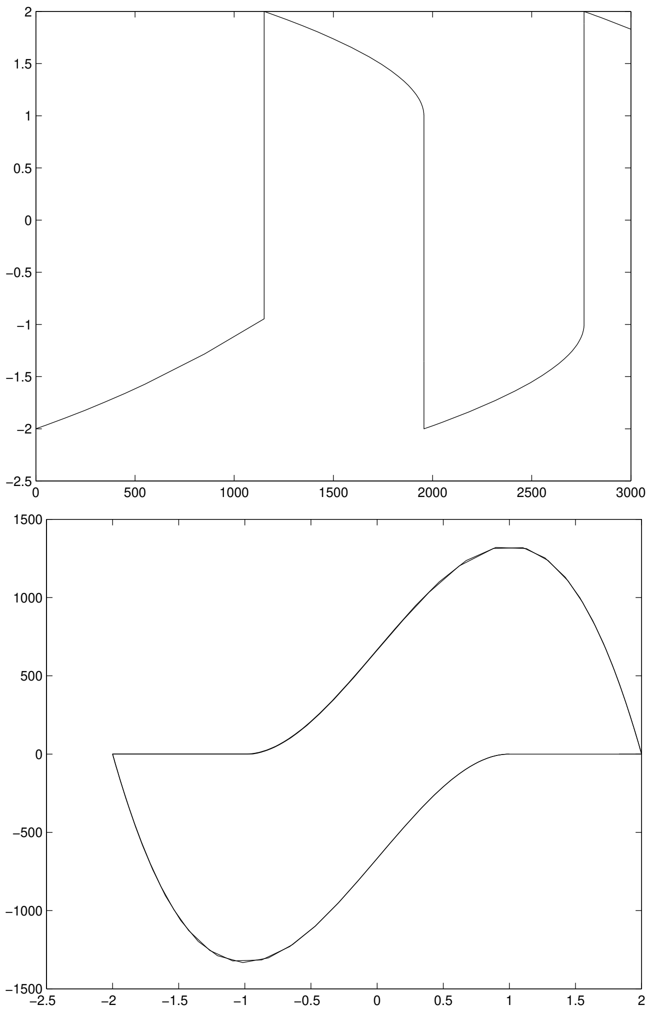

FIGURE 2: MATLAB plots of a solution to the van der Pol equation (2). Top: Time trace of x. Bottom: Phase plane.

Rather than define individual .m-files for the different versions of the van der Pol equation, it is convenient to consider a parametrised family

$$\dot{x} = y, \;\; \dot{y} = \mu(1-x^2)y - x, \tag{3}$$

and then afterwards specify the relevant values of the parameter μ. Parameters may be transported as follows. Define the file vdppar.m with the following contents

```
function xdot = vdppar(t,x,flags,mu)
xdot = zeros(2,1);
xdot(1) = x(2);
xdot(2) = mu*(1-x(1)^2)*x(2) - x(1);
```

(The variable flags is a dummy variable which is not important here.) To obtain a solution with $\mu = 1$, type

```
>> [t,x] = ode45('vdppar', [0 20], [1 0], [ ], 1);
```

To draw a phase portrait you will need more than just one solution. Here is a primitive procedure in a file phasport.m that provides this easily:

```
function phasport(equations,timespan,plotrange,solver)
clf;
axis(plotrange);
hold on;
button = 1;
while button == 1,
[xinit(1),xinit(2),button] = ginput(1);
if button ~= 1 break; end;
[T,Y] = feval(solver,equations,timespan,xinit);
plot(Y(:,1),Y(:,2));
[T,Y] = feval(solver,equations,-timespan,xinit);
plot(Y(:,1),Y(:,2));
end;
```

In the procedure equations is the name of the .m-file defining the equations, timespan is the time interval wanted for the solutions, plotrange is of the form [xmin xmax ymin ymax] and defines the plotting window in the phase plane, and solver is the name of a MATLAB differential equation solver.

When called, a plotting window opens, and the cursor changes into a cross-hair. Clicking with the *left* mouse button at a point in the phase space gives the orbit through that point. First the equations are integrated forwards in time and this part of the orbit is plotted. Then the same is done backwards in time. In either time direction the orbit may become unbounded, and a warning may be printed in the MATLAB command window. This is generally of no consequence, but if you have chosen timespan to large, computations may be unacceptably slow. Then press "control-c" in the MATLAB command window to break the execution and adjust the time interval. When an orbit is drawn you can continue clicking at points, and when you are satisfied click any other mouse button in the window, and the procedure terminates.

Figure 3 is created by the call

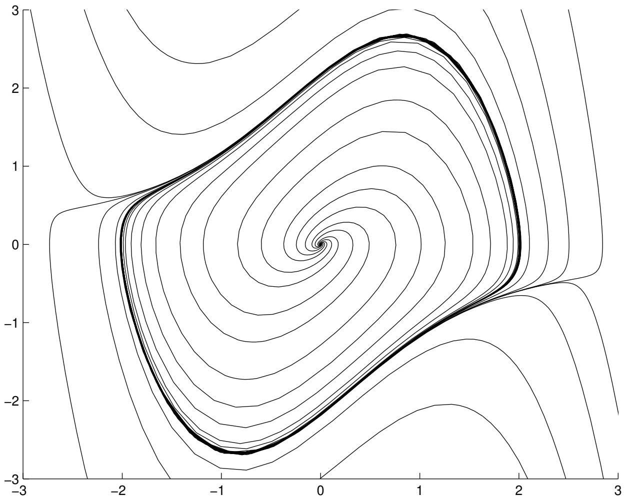

FIGURE 3: Phase portrait of (1) using the procedure `phasport`.

```
>> phasport('vdp1',[0 30], [-3 3 -3 3], 'ode45')
```

Again, you may want to include a parameter. This is done by `phasportpar.m`:

```
function phasportpar(equations,timespan,plotrange,solver,parameter)
clf;
axis(plotrange);
hold on;
button = 1;
while button == 1,
[xinit(1),xinit(2),button] = ginput(1);
if button ~= 1 break; end;
[T,Y] = feval(solver,equations,timespan,xinit,[ ],parameter);
plot(Y(:,1),Y(:,2));
[T,Y] = feval(solver,equations,-timespan,xinit,[ ],parameter);
plot(Y(:,1),Y(:,2));
end;
```

The phase portrait in figure 3 can then be also be obtained by the call

```
>> phasportpar('vdppar',[0 30], [-3 3 -3 3], 'ode45',1)
```